FireAxe: Partitioned FPGA-Accelerated Simulation of Large-Scale RTL Designs

Joonho Whangbo
UC Berkeley
joonho.whangbo@berkeley.edu

Edwin Lim
CMU
eglim@andrew.cmu.edu

Chengyi Lux Zhang
UC Berkeley
iansseijelly@berkeley.edu

Kevin Anderson
UC Berkeley
kevinand@berkeley.edu

Abraham Gonzalez
UC Berkeley
abe.gonzalez@berkeley.edu

Raghav Gupta
UC Berkeley
raghavgupta@berkeley.edu

Nivedha Krishnakumar
UC Berkeley
nivedha@berkeley.edu

Sagar Karandikar
UC Berkeley
sagark@eecs.berkeley.edu

Borivoje Nikolić
UC Berkeley
bora@eecs.berkeley.edu

Yakun Sophia Shao
UC Berkeley
ysshao@berkeley.edu

Krste Asanović
UC Berkeley
krste@berkeley.edu

***Abstract*—Pre-silicon validation and end-to-end system evaluation are integral parts of hardware development as they provide architects with insights about the complex interactions between various hardware components, system software, and application code. Although this process can be accelerated using FPGAs as a simulation host, existing platforms fall short when the resource requirements of a custom hardware design exceed a single FPGA.**

We present FireAxe, an open-source FPGA-accelerated RTL simulation platform that supports push-button user-guided partitioning across multiple FPGAs, using a compiler called FireRipper. Given a partition point, FireRipper automatically maps a monolithic RTL design onto multiple FPGAs while providing hardware designers quick feedback about the partition interface and expected simulation performance. Furthermore, FireRipper enables users to choose between an *exact-mode* which provides *cycle-exact* results with RTL-level fidelity, or a *fast-mode* that improves simulation rate while sacrificing fidelity only at the partition boundary. Built on FireSim, FireAxe preserves the ability to elastically scale simulations from on-premises FPGAs to cloud FPGAs. For example, pulling out a core from a system-on-chip (SoC) onto a separate FPGA, we achieve simulation rates of 1.6 MHz using on-premises FPGAs connected by direct-attach cables and 1 MHz on AWS F1 FPGAs using peer-to-peer PCIe.

To show FireAxe's ability to enable pre-silicon performance validation at unprecedented scale, we show several case studies. First, we replicate full-stack system-level effects such as latency spikes from garbage collection in a Golang application on an SoC containing 4 out-of-order (OoO) cores. We also boot Linux on, to our knowledge, the largest OoO core ever cycle-exactly simulated in academia. Lastly, we simulate a system-on-chip containing 24 OoO cores mapped onto five datacenter-class FPGAs. We discover an RTL bug when trying to run Linux user-space applications that did not appear with less substantial software stacks. This was discovered in less than 2 hours using FireAxe and would have taken weeks in a commercial software RTL simulator.

***Index Terms*—computer simulation, RTL simulation, FPGAs, performance analysis, computer architecture, scalability**

I. Introduction

The ever increasing demands of sophisticated applications from the edge to the cloud is driving the design of increasingly complex systems with larger die sizes in advanced technologies. Designing hardware at this scale is challenging due to the long turnaround times of capturing end-to-end system-level performance metrics, among many other design verification and validation tasks. While performance evaluation of a single module can quickly provide performance results in isolation, system-level performance validation, where the module is added into the full design, is required to measure realistic performance metrics.

Traditionally, abstract simulators have addressed this problem by providing a balance between simulation performance, fidelity, and flexibility. However, using abstract simulators is challenging due to the increasing amount of specialization in modern hardware designs. This is due to the fact that abstract simulators require validation, which is challenging without an RTL source-of-truth. Given an RTL source of truth, previous work such as FireSim [19] has enabled cycle-exact[1] simulation at 10s to 100s of MHz. This enabled architects to run realistic benchmarks and obtain high-fidelity results but requires that the design fits in a single FPGA.

In this work, we present FireAxe, an open-source[2] FPGA-accelerated RTL simulation framework built on FireSim that performs push-button user-guided partitioning of arbitrary hardware designs (e.g., a system on chip) across multiple FPGAs, using a compiler called FireRipper. FireRipper automatically partitions the design by extracting the user-specified modules into separate FPGA partitions while providing users quick feedback about the partition interface and expected simulation performance. In addition, it supports two modes that users can choose from: an *exact-mode* that provides *cycle-exact* simulation results with full RTL-level fidelity, or a *fast-*

[1]Cycle-exact means that the cycle numbers obtained by FireSim will be identical to the numbers obtained by running the same workload on a taped-out chip built from the same RTL.

[2]This will be open-source/upstreamed to FireSim (docs: https://fires.im/ github: https://github.com/firesim/firesim).

mode that provides higher simulation performance by sacrificing simulation fidelity only at the partition boundary. FireAxe supports both on-premises and cloud FPGA platforms such as AWS EC2 F1, preserving FireSim's ability to run large-scale simulations without paying the upfront cost of purchasing FPGAs. On AWS EC2 F1 instances, the communication between FPGAs occurs directly between FPGAs using peer-to-peer PCIe, reaching up to 1 MHz target simulation frequency. In on-premises settings, FPGA-to-FPGA communication is achieved by cheap ($\sim$\$25), off-the-shelf QSFP cables, which enables a target simulation frequency of up to 1.6 MHz. *Lastly, to the best of our knowledge, FireAxe is the first work to identify the synergy of multithreaded simulation and partitioning, which can be used to amortize the inter-FPGA communication latency in cases when there are duplicate modules within the target hardware.*

To demonstrate the capabilities of FireAxe, we perform a number of case studies. First, we simulate an SoC with 24 out-of-order (OoO) cores connected in a ring topology, by partitioning the SoC across five Xilinx Alveo U250 FPGAs [29] and find an RTL bug *three billion cycles* into the simulation. Simulating this SoC with FireAxe enabled us to discover the RTL bug in less than two hours. Attempting to find this bug in software RTL simulation would have been impractical, given that a commercial software RTL simulator would have taken weeks to reach the bug.

Next, to show the flexibility of FireRipper as well as the new microarchitectural design space that can be explored, we model an OoO core (synthesis area of 1.56mm^2 in a commercial 16nm technology) that does not fit in a single Xilinx Alveo U250 FPGA. We split the core in half, with the core's backend (register renaming, physical register file, execution units) and load-store-units on one FPGA while placing the core's frontend (instruction fetch, branch prediction, fetch buffer) and the memory subsystem on another FPGA. *To the best of our knowledge, this is the largest OoO core ever cycle-exactly simulated in academia.*

Finally, to demonstrate that FireAxe can be used to capture complex interactions between the application, system software, and hardware *pre-silicon*, we reproduce two effects that can only be observed in a full-system multi-core SoC simulation. To investigate the microarchitecture-level interactions between cores, caches, buses, and a network interface controller (NIC), we study how scaling the number of cores (up to 12) affects the leaky-DMA problem [15]. In addition, we reproduce latency spikes induced by garbage collection for the Go programming language [23] by simulating an SoC containing four OoO cores.

II. Background

FPGA-based simulators require decoupling the simulated target design's clock from the host-FPGA clock to obtain accurate performance results. In this section, we explain how LI-BDNs are used to achieve host-clock decoupling, and review FireSim [19], an FPGA-accelerated simulator that uses LI-BDNs to obtain cycle-exact simulation results.

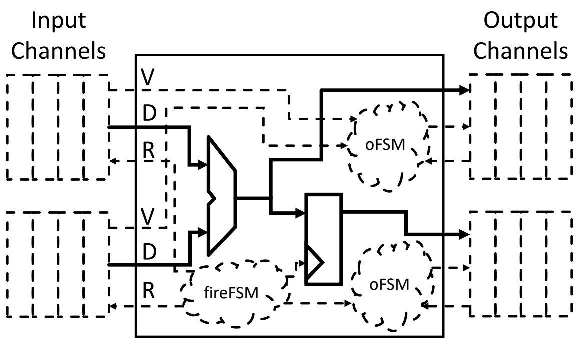


Fig. 1: LI-BDN. The solid lines indicate the original circuit while the dotted lines are extra circuitry added to generate the LI-BDN.

A. LI-BDNs

Directly emulating ASIC RTL on an FPGA leads to incorrect performance measurements due to the mismatch in FPGA frequency and taped-out ASIC frequency. For instance, an ASIC design targeting 1 GHz experiences 100 cycles of DRAM access latency when the backing DRAM latency is 100ns. However, if that unmodified RTL is mapped to an FPGA at 100 MHz with the same backing DRAM attached to the FPGA, then the DRAM access latency will be reduced to 10 cycles, resulting in inaccurate performance measurements. To solve this, latency-insensitive bounded dataflow networks (LI-BDN) [24] are used to decouple the target design's clock from the host FPGA clock by controlling when the target design can advance a cycle. From this point on, we refer to the ASIC RTL that is being simulated as the *target* design, while we refer to the platform that runs the simulation as the *host*.

Figure 1 is an LI-BDN where the solid lines represent the original target design and the dotted lines represent extra circuitry which controls when the target design can advance a cycle. The inputs and outputs of the target design interface with latency-insensitive channel queues that hold tokens or the data to be sent to/from the target design. Each output channel has a single-bit finite-state-machine (FSM) that waits until all the combinationally connected input channels have a valid token. When valid tokens are present, an output token is enqueued into the corresponding output channel. Furthermore, another FSM (`fireFSM` in Figure 1) advances a cycle when all the input channels have a valid token and all the outputs have fired or are firing. While advancing a cycle, all the input tokens are dequeued and the output FSMs are initialized.

B. FireSim

FireSim [19] is an example of applying this form of decoupling. FireSim is an open-source, FPGA-accelerated, cycle-exact hardware simulation platform that enables simulating RTL designs on FPGAs at 10s to 100s of MHz. FireSim contains a compiler called Golden Gate (GG) [20] that operates on FIRRTL [18]. FIRRTL is an intermediate representation (IR) for digital circuits. A FIRRTL description of a design is structured as an abstract syntax tree (AST), where nodes of the tree represent different elements in the digital circuit. By traversing the AST, the compiler can perform various operations such as modifying the design or performing static analysis.

The main transformation or compiler pass in GG, called FAME-1 [22], transforms the target design into an LI-BDN by attaching communication channels to the top-level I/O ports and adding FSMs to control when the target design can advance a cycle. Additionally, GG supports other optimizations to save FPGA area at the cost of simulation time. For example, another transform, called FAME-5, enables duplicate modules to be multi-threaded at the simulator level. For multi-threaded designs, combinational logic is shared across modules while sequential elements are replicated; the scheduler then selects the sequential elements to update for a certain cycle. Since FPGA LUT resources are a key resource-constraint when mapping ASIC designs onto FPGAs, multi-threading helps reduce the LUT utilization while increasing the utilization of relatively lesser-used BRAMs. In Section VI-B, we explain how we use FAME-5 to hide the communication latency between partitioned designs on multiple FPGAs.

III. FireRipper: FireAxe's Compiler

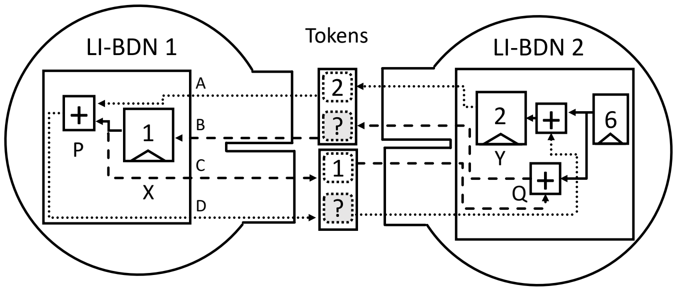
In this section, we present FireAxe's compiler, FireRipper, and the partitioning modes and module selection modes that it exposes when automatically constructing an FPGA-accelerated simulation. FireRipper is integrated as a part of GG and runs before the FAME transformations. Users can disable FireRipper to use the original FireSim flow, or enable it to perform partitioned simulations.

A. Partitioning Modes

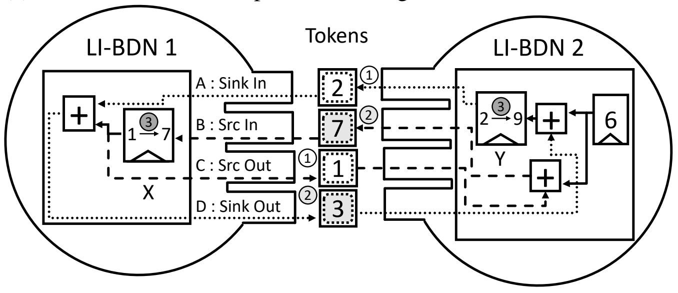
FireRipper supports two partitioning modes that allow the user to trade off simulation speed and simulation fidelity: *exact-mode* and *fast-mode*.

1) Exact-Mode: In the *exact-mode*, FireRipper partitions the SoC onto multiple FPGAs without modifying the target RTL to obtain a *cycle-exact* simulation with respect to the target RTL. One challenge is that when the partitioned interface contains combinational logic between the input and output ports, a token exchange must occur multiple times on separate input/output channels for the simulation to advance without deadlocking [9]. This is not an issue in the monolithic FireSim case as the combinational logic of the target is contained within a single LI-BDN, enabling the simulation to execute the logic in a single host FPGA cycle no matter how complicated the logic is.

Figure 2a depicts the case where all the input ports are concatenated and attached to a single LI-BDN channel and vice-versa for the output ports. This results in a deadlock due to the following reasons. For LI-BDN 1 to generate an output token, it needs to compute the output of adder P (port D) which is combinationally dependent on port A. Hence, it needs to wait until LI-BDN 2 sends a token containing the value for port A. For LI-BDN 2 to generate an output token, it also needs



(a) I/O channels are not separated, causing the simulation to deadlock.



(b) I/O channels are separated, enabling the simulation to make forward progress even though the partition boundary contains combinational logic.

Fig. 2: Exact-Mode. Modeling combinational logic between LI-BDNs requires multiple token exchanges on separate I/O channels to advance a single cycle without deadlocking.

to compute the output of adder Q which is combinationally dependent on port C. However, the token containing the value of port C cannot be sent until the value of port D is computed, resulting in a circular dependency between the tokens.

To avoid deadlocking the simulation, we first have to separate the I/O ports that have combinational dependencies (sink ports) from the ones that do not (source ports) as in Figure 2b. FireRipper automates this process by running multiple passes. First it topologically sorts the modules according to their position in the module hierarchy. Then it traverses the FIRRTL AST of each module identifying statements that are combinationally dependent to each other. Once this is done for a module, it can identify the output ports of the module that are combinationally dependent on its input ports. Now that FireRipper has differentiated the sink ports from the source ports, it concatenates all the input wires of the sink/source ports and attaches an LI-BDN input channel to the aggregated wires and vice-versa for output ports.

At this point, we can compose the LI-BDNs and start executing the simulation. The simulation makes forward progress like so:

1) During step 1, LI-BDN 1 generates an output token on the source out channel as it does not have any combinational dependencies to any of the input channels. The token value is `one` as it is the value of register X. Furthermore, assuming that the length of the combinational dependency chain between the input and output ports is less than or equal to two, it receives an

input token on the sink in channel. This is because the combinational dependency chain between the sink in and sink out ports is already two (port A to port D), implying that the sink in ports are all connected to sequential elements in LI-BDN 2. The token value is 2 as it is the value of register Y.

2) In step 2, LI-BDN 1 generates an output token on the sink out channel as the combinationally dependent sink in channel received an input token in the previous step. The value of the token is the output of the combinational logic which is 3. Similarly, LI-BDN 2 generates an output token of value 7 on the source in channel.
3) Finally in step 3, both LI-BDNs consumes the input tokens and they each update their register values to 7 and 9 respectively.
4) By repeating the above three steps, the simulation can run to termination.

As can be seen in the above example, by having separate LI-BDN channels for the source and sink ports, the initial seed token is generated on the source output channel by construction, thereby eliminating deadlocks. Furthermore, two token transfers were required to simulate a single target cycle.

One assumption that FireRipper makes in the above process is that the length of the combinational dependency chain between the input and output ports is less than or equal to two. This is because the length of the chain corresponds to the number of separate input/output channels as well as the number of inter-FPGA link crossings to simulate a single cycle. Hence, supporting these partitioning boundaries not only complicates the compiler, but will result in low simulation performance as the link has to be crossed multiple times to simulate a single cycle. When the length of the dependency chain is larger than 2—for example in a case where an output port A is combinationally dependent on an input port B which is again combinationally dependent on an output port C—FireRipper terminates compilation while providing the user with the chain of combinational ports that caused the termination.

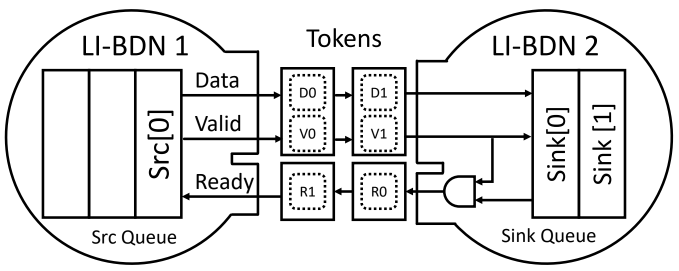
As discussed below, we cannot use *fast-mode* in cases where the partition boundary is not latency-insensitive. In these cases, we can utilize the *exact-mode* which supports higher partitioning flexibility. The *exact-mode* can also be used in cases where simulating the design with full fidelity is crucial: e.g., performance validation.

2) Fast-Mode: As mentioned above, simulating combinational logic between LI-BDNs placed on separate FPGAs requires multiple FPGA-to-FPGA link crossings to simulate a single target cycle, which can harm simulation performance. In FireAxe's *fast-mode*, we present a method to speed up simulation performance by nearly 2× over the *exact-mode*. The *fast-mode* is limited to partitioning boundaries that are latency-insensitive, and FireRipper no longer has to worry about the length of the combinational dependency chain as in the *exact-mode*.

Instead of differentiating the ports by whether they have combinational dependencies or not, in the *fast-mode*, FireRipper simply concatenates all the input ports and attaches an input LI-BDN channel to the concatenated wires and vice-versa for the output ports. However this can cause simulation deadlocks, as seen in Section III-A1, when there are combinational dependencies between the ports. This can be overcome by manually seeding each side of the LI-BDN with a single initial token. This prevents deadlocks as well as enabling each FPGA partition to run a single cycle in parallel before they produce an output token to be sent to the other side.

One problem with this approach is that inserting a seed token on all LI-BDNs at initialization time is equivalent to injecting a single cycle of latency between the partitioned interfaces. This is because an output token produced on one LI-BDN at cycle N will be seen at cycle $N + 1$ on the other LI-BDN since one token will always be in front of the token generated at cycle N. While this is not a problem for latency-insensitive interfaces that are credit based, this breaks the backpressure logic between ready-valid interfaces.

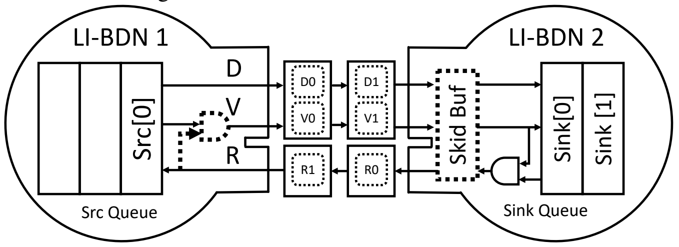
Figure 3a and Figure 3b shows where inserting a seed



(a) Fast-mode without module boundary modifications.

		Data Values								
		Src[0]	D0	V0	D1	V1	Sink[0]	Sink[1]	R0	R1
Simulation Step	1	A	-	-	Seed (X)	Seed (0)	Empty	Empty	-	Seed (0)
	2	A	A	1	-	-	Empty	Empty	0	-
	3	A	-	-	A	1	Empty	Empty	-	0
	4	A	A	1	-	-	A	Empty	1	-
	5	A	-	-	A	1	A	Empty	-	1
	6	Empty	F	0	-	-	A	A	0	-

(b) Step by step execution of Figure 3a. The y-axis indicates each step at which tokens are moved around, while the x-axis corresponds to each label in Figure 3a.



(c) Modifying the target boundary to preserve backpressure on a ready-valid interface.

Fig. 3: Fast-Mode. To preserve transactions going through the ready-valid interface, FireRipper inserts skid-buffers on the ready-valid sink side while lowering the valid signal until the received ready signal is high on the ready-valid source side.

token when partitioned across a ready-valid interface can break backpressure. Each row of Figure 3b corresponds to a simulation step while each column corresponds to the values in Figure 3a. Note that there is an extra token in between the LI-BDNs in Figure 3a compared to the *exact-mode* case to indicate that one cycle of latency is injected between the interfaces. The simulation progresses like so:

1) In step 1, we seed the input of LI-BDN 1 with a `R1` token with a value of `0`, and the input of LI-BDN 2 with a `(D1, V1)` token with a value of `(X, 0)`.
2) In step 2, LI-BDN 1 consumes the `R1` token and generates an output token where `(D0, V0)` is `(A, 1)` as `Src[0]` is a valid queue entry with a value of `A`. At the same time, LI-BDN 2 also consumes the `(D1, V1)` token and generates the `R0` token where the value is `zero` because the incoming `V1` is also a `zero`. Note that by seeding each side with an initial input token, both sides can simulate a single cycle in parallel without having to cross the inter-FPGA link multiple times, resulting in a higher simulation performance compared to the *exact-mode*.
3) In step 3, tokens `(D0, V0)` and `R0` are sent to the other side, becoming into the `(D1, V1)` and `R1` tokens respectively.
4) Steps 4, 5, and 6 are the repetition of steps 1, 2, 3 in Figure 3b, with the new values.

At step 6, we can see the backpressure logic breaking as the sink queue received two valid entries even though the source queue only had one.

To overcome the above problem, FireRipper transforms the target interface to sustain backpressure between LI-BDNs with latency in between as in Figure 3c. First, FireRipper inserts a skid buffer on the ready-valid sink side to prevent requests from getting lost. Next, FireRipper changes the valid signal to *valid & ready* on the ready-valid source side to prevent the source side from sending the same request multiple times. These two modifications enable the ready-valid interfaces to function correctly even with injected latency.

As a consequence of the initial token seeding and the target modifications, the simulation results are no longer *cycle-exact* with respect to the unmodified target-RTL and are only *cycle-approximate*. Nevertheless, as the modified target-RTL is also wrapped in an LI-BDN, the performance results obtained in the *fast-mode* are still *cycle-exact* with respect to the modified target-RTL resulting from these systematic transforms. We investigate the performance and accuracy tradeoffs of *fast-mode* in Section VI.

A compelling use case for *fast-mode* would be to partition modules that are attached to buses such as core tiles or MMIO devices as they interface with the bus via decoupled interfaces. The benefit is that these modules are normally coarse-grained enough such that they provide significant resource savings while having a narrow partition boundary width, which is beneficial to simulation performance.

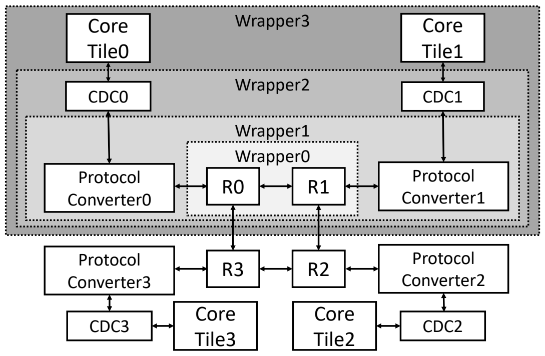


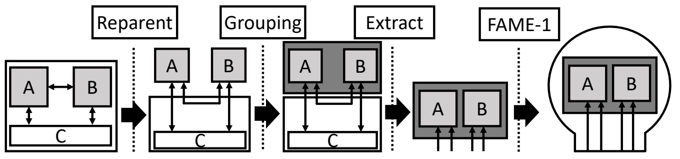
Fig. 4: NoC-partition-mode wrapper boundaries. FireRipper exploits the microarchitectural semantics of NoC router node boundaries and uses them as compiler hints.

B. Module Selection

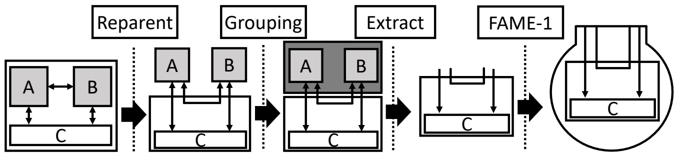
In addition to the partitioning mode, the user must provide the number of FPGAs to map the target-design onto and the set of modules that should be allocated to each partition. The default method for mapping modules onto each partition is to manually list out each module that should be put on each FPGA, giving users fine-grained control over module placement.

As an example of taking advantage of microarchitectural semantics to more efficiently simulate designs, FireRipper also allows users to divide modules across FPGAs using Network-on-Chip (NoC) boundaries. Since NoC router boundaries are credit based (i.e., latency-insensitive), and the output ports are not combinationally dependent on any of the input ports, these boundaries can be used as hints about where to partition a design. Figure 4 depicts an SoC where the bus is configured as a ring NoC topology. We use Constellation [32], a NoC generator that supports a wide range of topologies and routing schemes. The generated NoC from Constellation has three layers: the physical layer which contains the router nodes, the protocol layer which contains the protocol converters, and the top layer which contains the clock domain crossings (CDCs). To partition across NoC boundaries, instead of having to explicitly specify the set of modules to group together, FireRipper expects a set of router node indices that should be grouped together in a partition. Figure 4 describes how the NoC-partition-mode automatically selects the modules to partition out given a set of router node indices.

1) The user specifies the router nodes that should be partitioned out from an FPGA. These are `R0` and `R1` in Figure 4.
2) FireRipper automatically detects the neighboring router nodes by traversing the circuit representation, and draws boundaries between the NoC router nodes so that it can wrap the group of router nodes to partition in a wrapper module. This is `Wrapper0` in Figure 4.
3) FireRipper traverses the circuit representation again, collecting all the modules that are connected to the modules



(a) FireRipper extracts the user specified modules by first pulling them out to the top of the module hierarchy, wrapping them in one module, and deleting all the other modules.



(b) Similar to the module extraction transform, FireRipper reparents and wraps the modules to be removed. Once this is done, it deletes the wrapped module.

Fig. 5: FireRipper's main partitioning phase. Consists of transformations to remove and extract modules from the module hierarchy. The above example depicts how FireRipper partitions the design onto two FPGAs.

inside the wrapper module, but are not connected to any other modules. In the above figure, `Protocol Converter0` and `Protocol Converter1` are selected as they are connected to `Wrapper0` but not to any other router nodes. The selected modules, along with `Wrapper0` are wrapped once again into `Wrapper1`.

4) The previous step of collecting modules and wrapping them is repeated recursively, until the wrapper reaches the top of the module hierarchy. At this point, all the modules that need to be partitioned out are nicely grouped in a single wrapper module, `Wrapper3` in the above figure. The following FireRipper passes simply extract `Wrapper3` out of the module hierarchy to perform partitioning.

C. *Module Extraction and Removal*

Once modules are selected, FireRipper extracts the modules from the module hierarchy onto separate FPGA partitions as shown in Figure 5. Figure 5a depicts the transform that extracts the user specified modules (Modules `A` and `B` in the diagram) out of the module hierarchy and into their own partition. This consists of the following steps:

1) Reparent: Reparents the selected modules until they are positioned at the top of the module hierarchy. While reparenting, I/O ports are punched out as necessary. In Figure 5a, after the "Reparent' pass, we can see that the user specified modules `A` and `B` are pulled out of the top module while maintaining how their I/Os are connected with respect to the original module hierarchy.
2) Grouping: Groups the selected modules according to the number of FPGAs and wraps them in a wrapper module. In the above example, we are partitioning the design across 2 FPGAs and hence we wrap all the selected modules into one wrapper module. After the "Grouping" pass in Figure 5a, we can see that an extra module (dark grey box) is created that contains the user specified modules `A` and `B` as submodules.
3) Extract: Deletes all the modules except the wrapper module. After the "Extract" pass in Figure 5a, we can see that only the module containing module `C` is deleted, and only the wrapper module is left.
4) FAME-1: The extracted wrapper module is passed on to GG, which performs the FAME transformations. As GG is able to ingest any FIRRTL based module and transform it into an LI-BDN, no additional checking is required on the extracted wrapper module.

Similarly, Figure 5b depicts how the user specified modules are removed. All the steps are identical with the extraction case except that we delete the wrapper module and only maintain the rest of the module hierarchy. After modules are extracted or removed, the aforementioned timing mode modifications are applied before running the original FAME-1 LI-BDN transformation.

IV. FPGA-TO-FPGA TRANSPORT MECHANISMS

In this section, we elaborate on the various FPGA-to-FPGA transport mechanisms FireAxe uses to run on a variety of platforms: on-premises FPGAs and public cloud FPGAs like Amazon Web Services (AWS) EC2 F1.

A. *Host-Managed PCIe*

FireAxe can exchange tokens between FPGAs using a PCIe DMA interface through the host CPU. In FireAxe each FPGA partition has a corresponding C++ simulation driver running on the host CPU that has the FPGA attached. The driver can push tokens into the FPGA as well as pull tokens out from the FPGA through the FPGA's 512b-wide PCIe DMA interface. Once a driver pulls a token from an FPGA, it is sent to the receiving driver by writing to a shared memory region, which in turn pushes the token into its corresponding FPGA. The maximum simulation frequency is limited to 26.4 KHz due to software driver overhead and FPGA-to-host-CPU PCIe latency. However, as this does not require any special interconnect, it can be run on both cloud FPGA instances and on-premises FPGAs.

B. *Peer-to-peer PCIe*

To improve simulation performance on AWS EC2 F1 cloud FPGAs, we utilize their direct peer-to-peer inter-FPGA PCIe communication mechanism to reduce token exchange latency [7]. The *f1.16xlarge* and *f1.4xlarge* instances each contain multiple FPGAs (8 or 2 respectively) that can send and receive AXI4 [28] transactions directly to/from one another without going through the host. This provides the simulator up to 1 MHz target frequency.

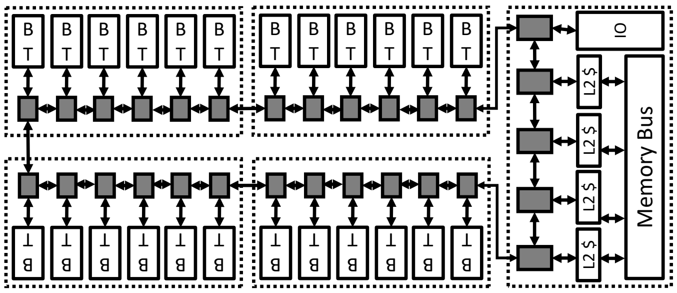


Fig. 6: 24 core SoC partition. We partition an SoC containing 24 BOOM tiles (BT) onto 5 FPGAs by utilizing the *NoC-partition-mode*. The entire simulation runs at 0.58 MHz.

C. *Off-the-shelf QSFP Direct-Attach Interconnects*

For on-premises FPGAs, we achieve an even lower link latency by utilizing cheap (∼$25), off-the-shelf QSFP direct-attach-cables [5] and integrating IP for the Aurora protocol [27] into the FPGA shell. This exposes an AXI4-Stream [28] interface to FireAxe. This ultra-low-latency interconnect enabled us to achieve a target simulation frequency of 1.6 MHz. Simulation performance of different hardware platforms is discussed in more detail in Section VI.

V. Case Studies

In this section, to show FireAxe's ability to support large-scale SoC simulations across multiple FPGAs and the new research opportunities that it enables, we perform several case studies. Throughout this section, we use our local FPGA cluster consisting of six Xilinx Alveo U250s [29] for evaluation.

A. *Simulating a 24 Core SoC*

As a demonstration of the simulation scale that FireAxe enables, we simulate an SoC containing 24 OoO cores (BOOM cores) [33] split across 5 FPGAs. As Figure 6 depicts, the cores are connected as a ring topology where we use the *NoC-partitioning-mode* to split the SoC across the NoC boundaries. We place 6 BOOM tiles (a tile consists of a core, L1-I, L1-D, and a crossbar) on four FPGAs and the SoC subsystem on the last FPGA. The BOOM tiles are multi-threaded using the FAME-5 transformation to save FPGA LUT resources.

We were able to boot Linux successfully on this SoC and run various small binaries. However, after adding larger binaries into the disk image via file system overlays, an RTL bug manifested while loading data from the disk, causing a supervisor binary interface trap after *3 billion cycles* into the simulation. To make sure that this is an RTL bug in BOOM, we swapped the BOOM cores to in-order cores (the rest of the SoC subsystem was identically configured) and retried booting Linux with the same disk image. In this case, we were able to boot Linux and successfully execute the larger binaries, indicating an issue in the BOOM RTL. Running this SoC at 0.58 MHz enabled us to discover this previously unknown RTL bug in less than 2 hours. The same SoC design ran at 1.26 KHz in a commercial software RTL simulator (giving FireAxe a 460× speedup), which translates to weeks of simulation time to trigger the same bug. We are currently in the process of identifying the RTL bug.

	Large BOOM	GC40 BOOM	GC Xeon
Issue width	3	6	6
ROB entries	96	216	512
I-Phys Regs	100	115	280
F-Phys Regs	96	132	332
Ld queue entries	24	76	192
St queue entries	24	45	114
Fetch buffer entries	24	54	144
L1-I	32 kB	32 kB	32 kB
L1-D	32 kB	32 kB	48 kB

TABLE I: Major microarchitectural parameters across Large BOOM, Golden-Cove-like BOOM (GC40 BOOM), and Golden Cove Xeon (GC Xeon)

B. *Simulating a Large-Scale Out of Order Core*

To show that FireAxe opens up new microarchitectural design space exploration opportunities, we split a large OoO core that does not fit on a single FPGA in half and place it on two FPGAs. At the same time, this partition point shows FireRipper's capability of partitioning a design across a wide range of user specified module boundaries as the core contains many cross-module signals. We downsize the microarchitectural parameters of Intel's Golden Cove (GC) [3], [11], [16] architecture by 40% and apply those parameters to BOOM. We call this configuration of BOOM the GC40 BOOM.

The microarchitectural parameters of the cores are shown in Table I. Furthermore, to provide a sense of how large these cores are, we obtain ASIC area estimations of the BOOM configurations by synthesizing the designs in a commercial 16nm process and compare them to published Xeon area numbers [4]. To perform an apples to apples comparison, we report the area of the core and the L1 caches for both Xeons and BOOM variants. The area of Large BOOM, GC40 BOOM and Xeons are each 0.79mm^2, 1.56mm^2, and 9.13mm^2 respectively. The numbers suggest that there is significant room for microarchitectural innovation in open-source OoO core design, and this innovation will be enabled by FireAxe's ability to model these growing cores by partitioning their simulation across multiple FPGAs.

When building the bitstream monolithically for GC40 BOOM at 10MHz without partitioning, the bitstream build fails due to congestion. Consequently, to simulate GC40 BOOM, we partition the core's back-end (register renaming, physical register file, execution units) and load-store-units from the frontend (instruction fetch, branch prediction, fetch buffer) and memory subsystem using the *exact-mode*. The backend side of the design takes a total of 63% of the total FPGA LUTs while the frontend and L1 cache side of the target takes up 18% of the total FPGA LUTs and the number of bits going through the partition interface is over 7000 bits. By partitioning the core onto 2 FPGAs, we are able to boot Linux with a overall target simulation frequency of 0.2 MHz. *To the best of our knowledge, this is the largest OoO core*

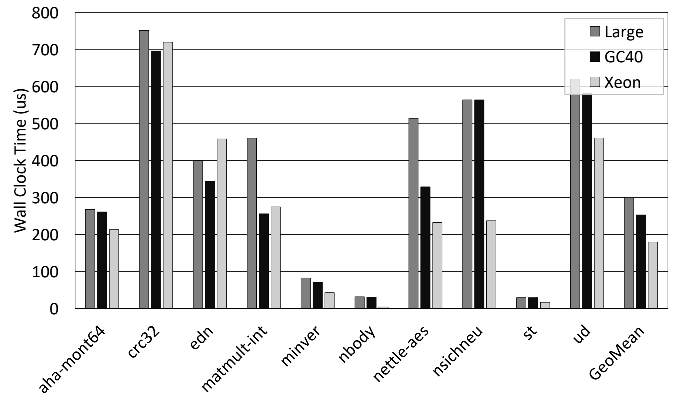


Fig. 7: Runtimes for Large BOOM, GC40 BOOM and Xeon running Embench workloads.

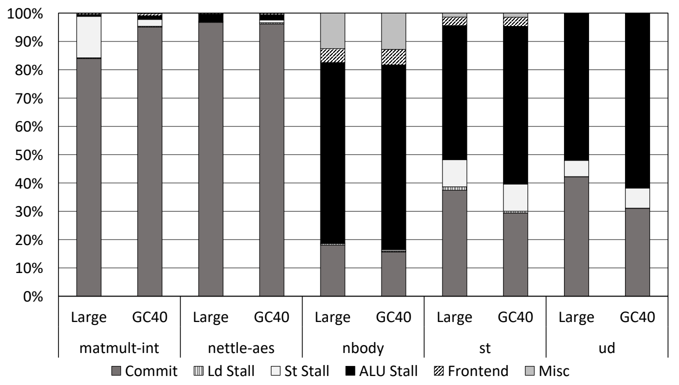


Fig. 8: CPI stack for Large BOOM and GC40 BOOM running Embench workloads.

to be simulated on an FPGA-based simulation platform in academia.

To understand how microarchitectural parameters affect core performance, we take a step further by comparing the runtime of Embench [8] between Large BOOM [33], GC40 BOOM and Xeons in Figure 7 (Large BOOM and GC40 BOOM were simulated while we ran Embench directly on Xeons). For our BOOM variants, we assume a clock frequency of 3.4 GHz as that was the frequency at which the Xeons were running when we ran Embench on them. We can see that GC40 BOOM consistently does well compared to Large BOOM with a 15.8% increase in average IPC.

Furthermore, we report CPI stacks for Large BOOM and GC40 BOOM by integrating a profiler called TIP [17] into FireAxe. Figure 8 shows where the core is spending its cycles for a selected set of benchmarks (these benchmarks were selected to cover a wide range of performance changes). For example, with *nettle-aes* we see that the instructions in the core spend most of its cycles committing while for *nbody* the instructions stall due to pipeline hazards. These trends are reflected in the performance of each benchmark as GC40 BOOM achieves a 56% performance boost over Large BOOM for *nettle-aes* while achieving only 2% improvements for *nbody*. This is due to the fact that LargeBOOM's throughput is bound by its instruction fetch bandwidth for *nettle-aes* and increasing the frontend width by 2x significantly improves performance. For *nbody*, increasing the instruction fetch bandwidth or the OoO window does not help as most instructions are bound by the execution unit's throughput.

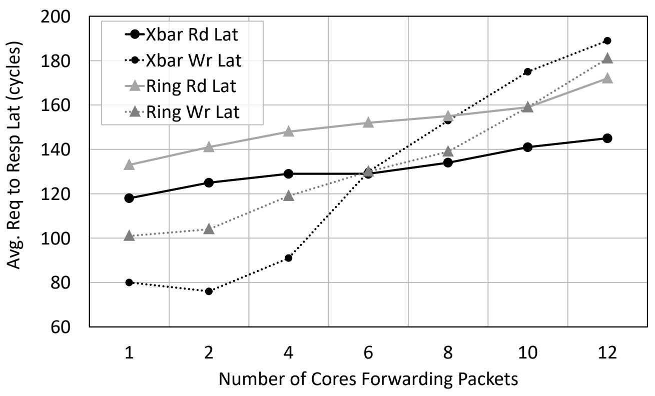


Fig. 9: Leaky-DMA effect. The Y axis shows the average request to response latency of each bus transaction. The read latency (Rd Lat) is for when the NIC reads the TX packets from the L2, while the write latency (Wr Lat) is for when the NIC writes the RX packets into the L2. We see that as we scale the number of cores, the average access latency goes up due to cache and bus contention.

C. Investigating the Leaky-DMA Effect in Server SoCs

As a case study of FireAxe, we investigate how scaling the number of cores results in the leaky-DMA effect, which has been identified as a problem in modern datacenter servers [15], [30]. Reproducing this effect requires simulating a large number of OoO cores, which would have been impossible without FireAxe's ability to perform large scale SoC simulations.

To reduce the latency between I/O devices and core, modern servers support a feature called data direct I/O (DDIO) which dedicates a configurable amount of last level cache (LLC) ways to IO devices, enabling them to read or write data directly into the LLC instead of DRAM. NICs use DDIO to inject packets into the LLC to reduce the core's packet access latency. However, there are cases when the processor doesn't benefit from DDIO. Specifically, when the size of the buffers used to interact between the core and the NIC is larger than the LLC portion dedicated to DDIO, incoming packets will evict cache lines containing packets that have not been processed by the core. Consequently, cache lines ping-pong between the LLC, DRAM and the core, a phenomenon called the leaky-DMA problem [15].

To reproduce this effect, we modify our NIC [19] such that it has a TX/RX queue corresponding to each core [25], enabling each core to interact with the NIC independently. Furthermore, we add hardware counters inside the NIC to measure the average bus request to response latency targeting the LLC and

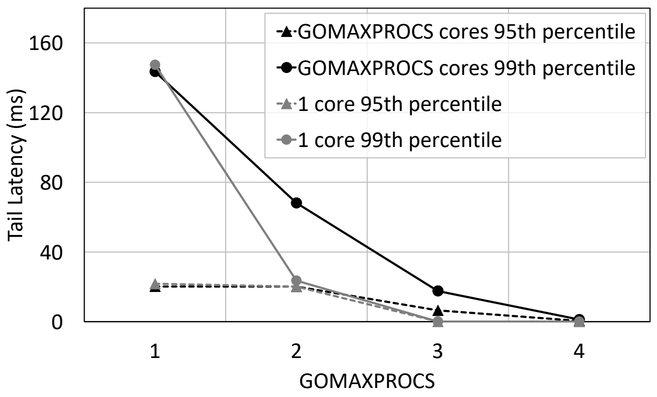


Fig. 10: Garbage collection tail latency. When GOMAXPROCS is set to one, we observe a high tail latency as the main goroutine is executed serially with the garbage collection goroutine.

use that as the proxy for cache hit rates. To resemble the LLC portion allocated to DDIO (2 cache ways), we resize the L2 cache to 128kB (in our systems the L2 is the LLC). The load is generated from the client SoC containing 12 in-order cores placed on a single FPGA and all cores are used to drive the NIC to send 1500B packets. The server SoC contains 12 BOOM cores split across 3 FPGAs and it simply forwards the incoming packets back to the client SoC. We measure the effects of cache and bus contention on the server SoC side by varying the number of cores that are forwarding the packets while dedicating a separate descriptor queue with 128 entries for each core. Lastly, we perform comparisons between two bus topologies: the cross bar based NoC (Xbar) and a NoC where the routers are connected as a bidirectional torus with a shortest path routing scheme (Ring).

We report the average request to response latency of each bus transaction in Figure 9. The latency is measured from the perspective of the NIC. That is, the read latency is the time it takes for the NIC to read the TX packets from the L2, while the write latency is the time for the NIC to write RX packets into the L2. We can see the read and write latencies increasing as the number of cores forwarding packets increase. This is due to the increased memory footprint of the packet buffers, resulting in cache contention on the limited DDIO ways. Not only that, we can see that the write latency of the cross bar bus (XBar) increases much more quickly than the Ring bus topology, resulting in a longer latency when scaling up to more than 6 cores. This shows that a NoC has a higher per bus transaction overhead compared to a cross-bar under low load, but it scales better under higher load.

D. Replicating Latency Spikes Induced by Garbage Collection in a Multi-core SoC

In this subsection, we replicate the latency spikes induced by garbage collection for the Go programming language (Golang). Replicating this effect requires full stack system level interactions between the user application, OS and hardware. The target SoC that we simulate contains four 5-wide OoO BOOM cores (where each takes up 25% of the Xilinx U250 LUT resources), and we split it across 2 FPGAs.

As the benchmark, we run a Golang application where the main goroutine is triggered by a periodic 10 microsecond tick [23]. This goroutine allocates objects in the heap recursively to stress the garbage collector. In this application, the tick as well as the main goroutine is delayed sporadically with respect to the 10 microsecond reference timer, and we measure the tail latency of this delay. We change the number of OS threads allocated to this Golang application via a flag exposed by the Golang runtime (GOMAXPROCS) [2] to see how the Golang runtime can exploit the extra threads. Additionally, we vary the CPU affinity such that the Linux scheduler can choose to run the threads on either one core or GOMAXPROCS cores. We report the 95% and 99% latency in Figure 10.

First of all, we can see that the 99% tail latency is very high when GOMAXPROCS is set to one. This is due to the fact that when there is only one OS thread allocated to this application, all the goroutines including the garbage collection goroutine are executed serially within a single thread context, delaying the execution of the main goroutine. On the other hand, when multiple OS threads are allocated to the Golang runtime, the runtime can distribute the goroutines to multiple OS threads which are scheduled by the Linux scheduler. Consequently, instead of waiting for all the previous goroutines to finish executing, the main goroutine can rely on the Linux scheduler to schedule the thread that it is mapped onto to start executing.

A surprising result is that the tail latency is lower when we pin this application to a single core rather than providing the Linux scheduler a pool of cores to run the application on. We hypothesize that, when running this benchmark on our system, the benefit of having higher cache affinity by running all the OS threads in a single core is larger than the benefit of running the OS threads across multiple cores while suffering from cache coherency.

To support our hypothesis, we run the same benchmark on our Xeon servers with GOMAXPROCS set to 2. When we allocate 2 cores from the same NUMA node, the 99% latency is 28 ms. In contrast, when allocating 2 cores from different NUMA nodes to exaggerate the inter-core communication latency, the 99% latency goes up to 42 ms. This shows that the inter-core communication overhead can increase the tail latency, and thus corroborates our suspicion that having a weak memory subsystem with a high cache coherency overhead can lead to the above effect. Nevertheless, we leave deeper investigation for future work.

VI. Simulation Performance and Validation

Understanding the throughput of a simulator is crucial as it is tightly coupled to the design iteration cycle. In this section we perform various sweeps to better understand the performance characteristics of FireAxe.

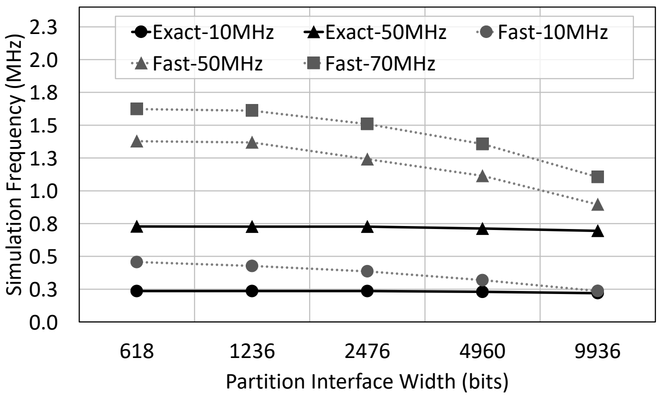


Fig. 11: QSFP performance sweeps. Simulation performance of FPGAs communicating via QSFP direct-attach-cables according to the bitstream frequency, partition interface width, and partition mode.

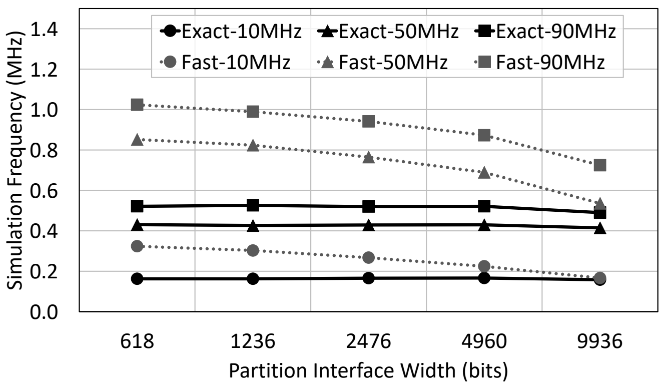


Fig. 12: PCIe peer-to-peer performance sweeps. Simulation performance of FPGAs communicating via PCIe peer-to-peer, according to the bitstream frequency, partition interface width, and partition mode.

A. Simulation Performance Characteristics

There are four major knobs that affect the performance of FireAxe. We elaborate on how each knob affects performance.

- Interconnect : A high bandwidth, low latency interconnect improves communication performance.
- Partitioning mode : This affects how may times a token has to cross the interconnect to simulate a target cycle.
- Module selection : The module selection dictates the width of the partitioned boundary. As the partition interface gets wider, the performance goes down. This is because the overhead of having to (de)serialize the bits to send and receive over the inter-FPGA communication link scales with the interface width.
- Bitstream frequency : As the bitstream frequency goes up, the performance improves as it takes less time for the simulator to produce/consume tokens. Furthermore, the (de)serialization overhead decreases since that logic is implemented as a part of the bitstream.

The complexity of the target logic along the partition boundary does not affect performance as long as the compiler requirements for each mode is met : i.e., limited combinational dependency chain length in the *exact-mode*, and latency insensitive boundaries in the *fast-mode*. We present a comprehensive set of performance sweeps in the following sections.

1) Partitioning Across On-premises FPGAs with QSFP Direct-Attach-Cables: First, we sweep over three variables that affect simulation performance when partitioning the design across two FPGAs: the number of bits going through the interface, the frequency at which the target bitstreams were built, and the compiler mode by which FireRipper performed the partition. The interface width is varied by changing the number of core tiles that are partitioned out of the SoC. Note that the SoC in this experiment is a bus based design (core tiles are connected by a cross-bar) and we are using the default module selection mode and *not the NoC-partition-mode* to pull out the core tiles. The inter-FPGA communication is performed by QSFP direct-attach-cables.

Figure 11 shows the results of the performance sweeps. There are two major overheads when transporting tokens between FPGAs: the inter-FPGA communication latency and the overhead of (de)serializing the AXI-Stream data exposed by the Aurora IP into simulator tokens. When the simulator is partitioned by the *exact-mode*, the inter-FPGA communication latency dominates the (de)serialization overhead as tokens have to traverse the inter-FPGA link twice. Similarly for the *fast-mode*, the inter-FPGA communication latency dominates for partition interface widths smaller than 1500 bits, providing us with around a 2x speedup over the *exact-mode*. However, once the partition interface gets wider than 1500 bits, the performance benefits of *fast-mode* becomes marginal as the (de)serialization overhead starts becoming on par with the inter-FPGA communication latency.

2) Partitioning Across Cloud FPGAs with Peer-to-Peer PCIe: We report the performance of FireAxe in Figure 12 when the design is split across 2 FPGAs communicating via the PCIe peer-to-peer [7] communication scheme, while sweeping over the same variables as in VI-A1. The overall performance characteristics are similar with the on-premises FPGA setup: *exact-mode's* performance stays relatively stable, *fast-mode* gives around a 2x performance boost over *exact-mode* until the interface width becomes wide enough such that the (de)serialization overhead becomes significant. Overall, FireAxe's performance on the cloud is 1.5x lower than on the local FPGA setup due to the higher inter-FPGA communication latency.

3) Partitioning Across More than 2 FPGAs on Local FPGAs: We also perform sweeps to understand how the number of FPGAs connected in a ring affects the *NoC-partition-mode*'s performance in a local FPGA setting. The width of the interface stays constant as we are partitioning across the NoC router boundaries. We report the performance trends in Figure 13. As can be seen, even though each FPGA is

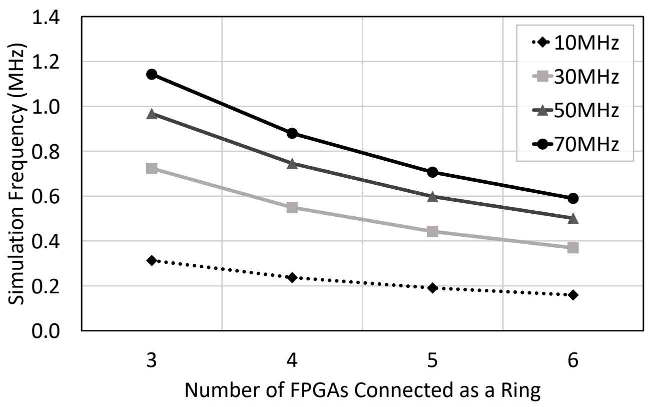


Fig. 13: FPGA count performance sweeps. Simulation performance according to the number of FPGAs connected as a ring for a particular target bitstream frequency.

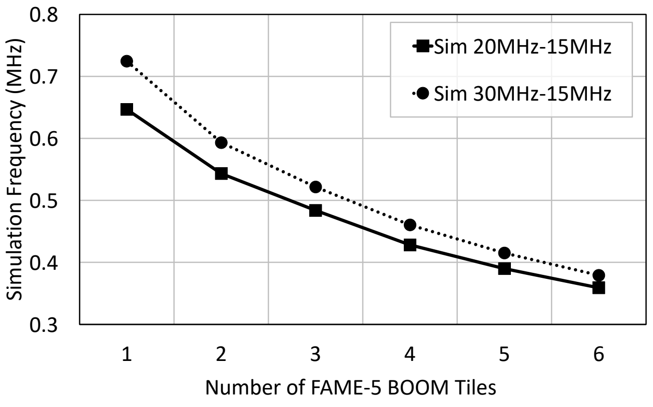


Fig. 14: Amortizing communication via FAME-5. Measures simulator performance according to the number of BOOM tiles that are multi-threaded. As we increase the number of tiles from one to six, the simulation performance degrades by less than 2x. This is a conservative estimate as the number of bits going off the FPGA increases linearly with the number of tiles (threads), negatively affecting simulation performance.

exchanging tokens only with its neighboring FPGAs in the ring connection, the performance drops as the number of FPGAs in the ring increases due to minor timing issues regarding token exchange.

B. *Hiding Communication Latency via FAME-5*

In this subsection, we explain how we amortize the communication costs between FPGA partitions by utilizing the FAME-5 transformation. FAME-5 [22] is a technique to multi-thread duplicate modules in the design. Given *N* duplicate module instances, it shares the combinational logic while the sequential elements are replicated *N* times. A hardware scheduler is added to select the sequential elements to update for a given cycle. This trades off simulation performance with FPGA resource consumption as the simulator has to spend *N* host FPGA cycles to simulate a single target cycle.

In FireAxe's case, we utilize the FAME-5 transformation to amortize the inter-FPGA communication latency between partitions. Since the inter-FPGA communication latency is significantly higher than the *N-1* extra cycles spent simulating a multi-threaded module, the communication penalty of simulating a large, multi-threaded design is almost identical as simulating a single module without any multi-threading. This implies that simulating a larger design with duplicate modules comes with near-zero overhead.

The understand the impact of FAME-5 on FireAxe's performance, we increase the size of the SoC by increasing the number of BOOM tiles. When there are N BOOM tiles in the SoC, we partition all of them out onto a separate FPGA and apply the FAME-5 transformation on all of those tiles. The simulation performance results are shown in Figure 14. We fix the BOOM tile side of the FPGA frequency to 15MHz while varying the SoC subsystem side of the FPGA frequency from 20MHz to 30MHz. *By increasing the number of tiles from one to six, we are simulating a larger design (more cores) while saving FPGA resource consumption, without significantly sacrificing simulation performance.* This type of scaling relationship is enabled by the ability to amortize inter-FPGA communication latency by FAME-5. Not only that, this is a conservative estimate as the number of bits going off the FPGA increases linearly with the number of tiles (threads), negatively affecting simulation performance.

C. *Simulator Validation*

As a validation of our simulator compared to a non-partitioned setup, we partition three different SoC setups using *exact-mode* and *fast-mode* and validate the runtime compared to its equivalent monolithic FireSim simulation. First we test two accelerator SoCs, one with an encryption accelerator (Sha3Accel), and another with a machine-learning accelerator (Gemmini), and separate each accelerator onto a separate FPGA. We run each accelerator SoC and measure the time it takes for the accelerator to perform the operation (encryption for Sha3Accel and a convolution operation for Gemmini). The final SoC configuration is partitioning a core tile out of an SoC (Rocket tile). We boot Linux on this SoC and terminate it immediately after it is booted and measure the total number of cycles from the start to the end of the simulation.

We report the cycle-count error with respect to each setup's monolithic FireSim simulations in Table II. The monolithic simulation and *exact-mode* simulation took the exact same number of cycles. On the other hand, the cycle-counts do not match in the *fast-mode* because of the target modifications made to improve simulation performance. This is due to the partition boundary modifications and the extra cycle of latency injected in between the boundaries. The error rate depends on the workload and the degree of which the module is sensitive to the latency in between the boundary. For example, Sha3Accel's workload is relatively small and memory latency bound which makes is more sensitive to the target modifications compared to Rocket and Gemmini.

	Monolithic (Cycles)	Exact-Mode |Error| (%)	Fast-Mode |Error| (%)
Rocket tile (Linux boot)	3840921346	No Error	0.98
Sha3Accel (Encryption)	302	No Error	6.62
Gemmini (Convolution)	4505	No Error	0.22

TABLE II: Simulator validation. Cycle-count comparisons with monolithic FPGA simulations vs simulations partitioned by *exact-mode* and the *fast-mode*.

VII. Related Work

In this section, we present previous tools that accelerate RTL simulation, and contrast them with FireAxe.

A. Industry Simulation Tools

ProtoCompiler [14] is a multi-FPGA partitioning compiler, targeting FPGAs. The Palladium Emulation [1] platform also has an accompanying compiler called Automated Parallel Partition Compile (PPC). Unlike FireAxe, which utilizes off-the-shelf hardware components or inexpensive pay-as-you-go cloud FPGAs, these industry solutions come at a substantial cost and are thus unsuitable for academic research, startups, or early-stage co-design. IBM has also demonstrated a multi-FPGA simulator where they compose LI-BDNs to simulate larger designs [6] like FireAxe. However, their methodology requires a custom FPGA platform and is not open-sourced for the community.

B. Academic FPGA-based Simulation Tools

FireSim is an FPGA-accelerated RTL simulator. Since the initial release of FireSim, the proliferation of open hardware IP has enabled researchers to generate new SoC configurations that no longer fit in a single FPGA. FireAxe builds on top of FireSim to enable partitioning of large designs onto multiple FPGAs, overcoming the single-FPGA limitation for monolithic RTL designs.

SMAPPIC [10] is an FPGA prototyping platform. It does not perform clock decoupling, which limits performance validation capabilities and results in non-deterministic modeling. Also, although SMAPPIC can place designs on multiple FPGAs, the partition points are limited to tile boundaries. On the other hand, FireAxe can partition designs across a wider range of module boundaries. MEG [31] is an FPGA-hosted simulator that focuses on the design space exploration of memory technologies. Unlike FireAxe, it is a single-FPGA simulation platform and does not automatically transform ASIC RTL into FPGA simulators.

C. Accelerating Performance of RTL Simulations

RepCut [26] proposes a novel circuit partitioning approach to efficiently parallelize software RTL simulations. They reduce the thread synchronization overhead by replicating nodes between partitions to remove intra-cycle dependencies, as well as by proposing a better prediction of each partition's execution time. Our work is orthogonal as we are partitioning our design because of FPGA resource constraints while they partition the design to extract thread level parallelism. However, there may be synergies in employing this automated partitioning approach in FireAxe in the future.

LiveSim [21] is an RTL simulator that drastically reduces the design iteration time. First, it uses incremental compilation to reduce the compile time, as well as checkpointing so that the simulator can verify if each region is reusable with the current design and skip to parts where results start diverging. Although LiveSim is useful during initial RTL development, it is still limited by the software RTL simulation speed, preventing it from running realistic benchmarks.

Recent work such as Elsabbagh et. al. [12] and Manticore [13] partition hardware designs via custom compilers and map them down to custom simulation ASICs to extract fine-grained parallelism inherent in RTL simulations. These works are orthogonal to FireAxe as FireAxe utilizes host-clock decoupling to directly map RTL designs onto widely-available FPGA platforms.

VIII. Discussion and Future Work

In this section, we suggest how FireAxe can be utilized for future research and suggest future avenues for further improving FireAxe's simulation capabilities.

A. Hybrid Cloud Usage Model of FireAxe

When deciding between cloud and on-premises FPGAs, three key factors stand out. Firstly, the cost: Cloud FPGAs charge by the hour, while on-premises setups require an upfront investment. Secondly, FPGA capacity affects design requirements, with our experience showing local Alveo U250s offering 50% more LUTs compared to cloud-based VU9Ps due to fixed IP in cloud FPGAs. Lastly, simulation performance varies, with on-premises setups offering better performance thanks to high-bandwidth, low-latency QSFP connections between FPGAs, as demonstrated in Sections VI-A1 and VI-A2.

Drawing from these observations, we advocate for a hybrid model encompassing both cloud and on-premises resources to facilitate efficient development utilizing FireAxe. Initially, during the developmental phase, users can leverage the on-premises setup to minimize simulation latency, enabling agile identification of functional and performance-related bugs within their designs. Subsequently, as the design progresses and there is a need to conduct a multitude of benchmarks to obtain performance metrics, users can seamlessly expand their simulations by deploying additional FPGA instances in the public cloud.

B. Further Automating the Partitioning Flow

In the future, the FireAxe flow can be augmented in such a way that it involves less user guidance. To support this feature, FireRipper would need to be able to make rough per-FPGA resource consumption estimates based on the RTL-level circuit representation to provide users quick feedback about whether the partition will fit on an FPGA or not. Using existing graph partitioning tools to automatically search for boundaries that are amenable to partitioning would be another possible direction in the future.

C. Supporting Diverse On-Premises FPGA Platforms and Topologies

In this work we primarily used the Xilinx Alveo U250 boards [29] as our on-premises FPGA platform. However, FireAxe is agnostic to the FPGA board as long as there is a common protocol between FPGAs that enables inter-FPGA communication. We leave the support of other on-premises FPGAs for future work.

Additionally, the on-premises FPGA connection topologies are currently limited by the number of QSFP cages available to directly connect FPGAs (the Xilinx Alveo U250 board supports two QSFP cages limiting the topology to a ring or binary tree-like structure). We can overcome this limit by exploring other inter-FPGA communication protocols such as Ethernet that support routing packets to any other FPGA via a central switch.

IX. Conclusion

To cope with the challenge of simulating designs with growing complexity, we built FireAxe, which serves as a push-button solution to map FireSim simulations of large monolithic RTL designs across multiple FPGAs, bypassing the capacity constraints of a single FPGA. FireAxe's compiler FireRipper, provides users with fine-grained control over partitioning granularity, as well as the ability to easily trade-off simulation performance and accuracy. To democratize access to partitioned FPGA simulations, FireAxe supports both public cloud FPGAs on AWS EC2 F1 and on-premises FPGA clusters with a range of FPGA-to-FPGA communication links. As shown throughout this work, FireAxe enables full system simulations of hardware designs with unprecedented scale and provides *high fidelity*, *high speed*, and *deterministic simulations*, serving as a highly accessible solution for running full-stack large-scale hardware simulations.

Acknowledgment

We would like to thank Vighnesh Iyer, Jerry Zhao, David Biancolin, Christopher Batten, and the anonymous reviewers and artifact evaluators for their valuable feedback. We would also like to thank Björn Gottschall for answering questions about TIP [17]. Research was partially funded by the U.S. Government under the DARPA RTML program (contract FA8650-20-2-7006) and NSF CCRI ENS Chipyard Award 2016662 and NSF 2303735: Pathways to Enable Open-Source Ecosystems (POSE) and SLICE Lab industrial sponsors and affiliates. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

Appendix

A. Abstract

This artifact appendix describes how to reproduce the case studies and performance sweeps of FireAxe from Section V and Section VI. All of this code will be upstreamed in the main FireSim github repository (**https://github.com/firesim/firesim**).

B. Artifact check-list (meta-information)

- **Hardware:** Server with at least four Xilinx U250 FPGAs connected by QSFP cables.
- **Metrics:** Simulation throughput of FireAxe and simulation results from the case studies.
- **Output:** Figures about the above metrics.
- **Experiments:** FireAxe simulations of various large scale SoC designs explained in Section V and Section VI.
- **How much disk space required?:** 300GB.
- **How much time is needed to prepare workflow?:** 1 hour (scripted installation).
- **How much time is needed to complete experiments?:** 40 hours (fully automated).
- **Publicly available:** Yes.
- **Code licenses:** Several, see download.
- **Archived:** https://zenodo.org/records/11005136

C. Description

1) How to access: The artifact consists of one main git repository (fireaxe-firesim) which can be found on Zenodo (**https://zenodo.org/records/11005136**). This repository is a fork of the FireSim simulation environment. The dependencies are automatically pulled when running the installation script for this repository.

2) Hardware dependencies: To reproduce the experiments we will be using a local FPGA server with four Xilinx U250s installed. These FPGAs are connected by QSFP direct attach cables in a particular topology (5 FPGAs are connected in a ring).

D. Installation

First we need to login to a FPGA server containing four Xilinx U250 FPGAs.

```
$ ssh ⟨user-name⟩@⟨fpga-server-name⟩
```

From this point forward, all commands should be run on the FPGA server machine.

Next, we will create a tmux session so that the processes are not killed on a ssh disconnection. Also we will create a directory where you will download the artifacts from Zenodo.

```
$ tmux
$ cd /scratch
$ mkdir /scratch/⟨user-name⟩
$ cd /scratch/⟨user-name⟩
```

Fetch the top-level repository from Zenodo.

```
$ export ZENODO=https://zenodo.org/records/11005136
$ curl -Ls -w %url_effective -o a $ZENODO > DL_url
$ wget $(cat DL_url)/files/fireaxe-firesim.zip
$ unzip fireaxe-firesim.zip
```

At this point, we need to install conda as FireSim depends on it to download dependencies. You can install miniconda by running the following commands:

```
$ cd fireaxe-firesim/fireaxe-scripts
Follow the instructions in the prompt
Set the installation directory to /scratch/⟨user-name⟩
$ sh Miniconda3-latest-Linux-x86_64.sh
$ conda install -n base conda-lock==1.4.0
```

Next, run the following commands which will initialize all the following dependencies and setup FireSim and Chipyard. We will be using pre-built bitstreams which will be downloaded in this step as well.

```
$ cd fireaxe-firesim
$ ./build-setup.sh --skip-validate
```

This step should take around an hour. Upon successful completion, it will print:

```
Setup complete!
```

Once this is complete, we will activate the conda environment that was created in the previous step:

```
$ source sourceme-manager.sh --skip-ssh-setup
```

Now, we will initialize the FireSim manager:

```
$ firesim managerinit --platform xilinx_alveo_u250
```

At this point, the installation of FireSim and Chipyard to run FireAxe is finished.

Finally we need to make sure that we can ssh into localhost without a password. To setup the ssh-keys, run the following commands:

```
$ cd ~/.ssh
$ ssh-keygen -t ed25519 -C "username.pem" -f username.pem
[create passphrase : press enter without adding a password]
$ cd ~/.ssh
$ cat username.pem.pub » authorized_keys
$ chmod 0600 authorized_keys
$ cd ~/.ssh
$ ssh-agent -s > AGENT_VARS
$ source AGENT_VARS
$ ssh-add username.pem
```

The following command should not prompt for a password nor fail:

```
$ ssh localhost
```

Now, logout from the localhost ssh session to return to the original environment:

```
$ logout
```

E. Experiment workflow

We will now run the full artifact evaluation script which will do the following for each set of experiments:

1) Compile the target software to run on the simulated designs.
2) Compile the host drivers that controls the advancement of the simulation.
3) Flash the FPGAs with pre-built bitstreams.
4) Run FireAxe simulations.
5) Process the simulation results and reproduce the plots in the paper.

Now, run the full artifact evaluation script:

```
$ cd fireaxe-scripts
$ ./run-ae-full.sh
```

This step should take around 40 hours. Upon successful completion, it will print

```
run-ae-full.sh complete!
```

F. Evaluation and expected results

The above steps will produce a set of plots generated by running run-ae-full.sh. The plots are located in the fireaxe-scripts/generated-plots directory. The raw results of these experiments are included in the fireaxe-scripts/perf-results and fireaxe-scripts/ddio-results directory.

1) Figure 7 : 'figure-7-wallclocktime.png'
2) Figure 8 : 'figure-8-cpistack.png'
3) Figure 9 : 'figure-9-ddio.png'
4) Figure 10 : 'figure-10-go-gc.png'
5) Figure 11 : 'figure-11-qsfp-perf-sweep.png'
6) Figure 14 : 'figure-14-fame5-perf-sweep.png'

We do not run the EC2 F1 performance sweeps Figure 12 as it is redundant to Figure 11. We also do not run the performance sweeps according to the FPGA count (Figure 13) and the ring-NoC configuration of the DDIO experiment (Figure 9) as it requires manually rewiring the FPGAs on the server. However, the bitstreams for the above two experiments are included in the downloaded Zenodo repository for testing.

The results of Figure 11 and 14 may look different the paper as the performance of the driver code running on the host machine can alter the overall throughput of FireAxe. However, the difference should not be significant.

G. Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html

REFERENCES

[1] "Cadence palladium z1 enterprise emulation platform," https://www.cadence.com/content/dam/cadence-www/global/en_US/documents/tools/system-design-verification/palladium-z1-ds.pdf.

[2] "Go-runtime-documentation," https://pkg.go.dev/runtime, accessed: 2023-11-20.

[3] "Golden cove," https://en.wikipedia.org/wiki/Golden_Cove, accessed: 2023-10-15.

[4] "Skylake (server) - microarchitectures - intel," https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(server), accessed: 2023-10-15.

[5] 10Gtek, "Qsfp28 passive copper cable assembly," https://www.10gtek.com/qsfp28dac, 2019, accessed: 2023-10-15.

[6] S. Asaad, R. Bellofatto, B. Brezzo, C. Haymes, M. Kapur, B. Parker, T. Roewer, P. Saha, T. Takken, and J. Tierno, "A cycle-accurate, cycle-reproducible multi-fpga system for accelerating multi-core processor simulation," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 153–162. [Online]. Available: https://doi.org/10.1145/2145694.2145720

[7] awslabs, "Using-pcie-peer2peer," https://github.com/awslabs/aws-fpga-app-notes/tree/master/Using-PCIe-Peer2Peer/README.md, 2021, accessed: 2023-10-15.

[8] J. Bennett, "Embench™: A modern embedded benchmark suite," https://www.embench.org/, accessed: 2023-10-15.

[9] D. Biancolin, "Automated, fpga-based hardware emulation of dynamic frequency scaling," Ph.D. dissertation, UC Berkeley, Spring 2021.

[10] G. Chirkov and D. Wentzlaff, "Smappic: Scalable multi-fpga architecture prototype platform in the cloud," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 733–746. [Online]. Available: https://doi.org/10.1145/3575693.3575753

[11] clamchowder, "Popping the hood on golden cove," https://chipsandcheese.com/2021/12/02/popping-the-hood-on-golden-cove/, accessed: 2023-10-15.

[12] F. Elsabbagh, S. Sheikhha, V. A. Ying, Q. M. Nguyen, J. S. Emer, and D. Sanchez, "Accelerating rtl simulation with hardware-software co-design," in *Symposium on Microarchitecture (MICRO'23)*, 2023.

[13] M. Emami, S. Kashani, K. Kamahori, M. S. Pourghannad, R. Raj, and J. R. Larus, "Manticore: Hardware-accelerated rtl simulation with static bulk-synchronous parallelism," 2023.

[14] B. Erickson, "Solving the asic prototype partition problem with synopsys protocompiler," Jul 2014.

[15] A. Farshin, A. Roozbeh, G. Q. M. Jr., and D. Kostić, "Reexamining direct cache access to optimize I/O intensive applications for multi-hundred-gigabit networks," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, Jul. 2020, pp. 673–689. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/farshin

[16] A. Frumusanu, "Golden cove microarchitecture (p-core) examined," https://www.anandtech.com/show/16881/a-deep-dive-into-intels-alder-lake-microarchitectures/3, accessed: 2023-10-15.

[17] B. Gottschall, L. Eeckhout, and M. Jahre, "Tip: Time-proportional instruction profiling," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 15–27. [Online]. Available: https://doi.org/10.1145/3466752.3480058

[18] A. Izraelevitz, J. Koenig, P. Li, R. Lin, A. Wang, A. Magyar, D. Kim, C. Schmidt, C. Markley, J. Lawson, and J. Bachrach, "Reusability is firrtl ground: Hardware construction languages, compiler frameworks, and transformations," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov 2017, pp. 209–216.

[19] S. Karandikar, H. Mao, D. Kim, D. Biancolin, A. Amid, D. Lee, N. Pemberton, E. Amaro, C. Schmidt, A. Chopra, Q. Huang, K. Kovacs, B. Nikolic, R. Katz, J. Bachrach, and K. Asanovic, "Firesim: Fpga-accelerated cycle-exact scale-out system simulation in the public cloud," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 29–42.

[20] A. Magyar, D. Biancolin, J. Koenig, S. Seshia, J. Bachrach, and K. Asanović, "Golden gate: Bridging the resource-efficiency gap between asics and fpga prototypes," in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2019, pp. 1–8.

[21] H. Skinner, R. Trapani Possignolo, S.-H. Wang, and J. Renau, "Livesim: A fast hot reload simulator for hdls," in *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2020, pp. 126–135.

[22] Z. Tan, A. Waterman, H. Cook, S. Bird, K. Asanović, and D. Patterson, "A case for fame: Fpga architecture model execution," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 290–301. [Online]. Available: https://doi.org/10.1145/1815961.1815999

[23] tarm, "runtime: Gc causes latency spikes with single processor," https://github.com/golang/go/issues/18534, 2017, accessed: 2023-10-15.

[24] M. Vijayaraghavan and Arvind, "Bounded dataflow networks and latency-insensitive circuits," in *2009 7th IEEE/ACM International Conference on Formal Methods and Models for Co-Design*, 2009, pp. 171–180.

[25] A. Viviano, "Introduction to receive side scaling," https://learn.microsoft.com/en-us/windows-hardware/drivers/network/introduction-to-receive-side-scaling, 2022, accessed: 2023-10-15.

[26] H. Wang and S. Beamer, "Repcut: Superlinear parallel rtl simulation with replication-aided partitioning," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 572–585. [Online]. Available: https://doi.org/10.1145/3582016.3582034

[27] Xilinx, "Logicore ip aurora 64b/66b," 2011, accessed: 2023-10-15.

[28] Xilinx, "Axi reference guide," 2017, accessed: 2023-10-15.

[29] Xilinx, "Alveo u200 and u250 data center accelerator cards data sheet (ds962)," 2023, accessed: 2023-10-15.

[30] Y. Yuan, M. Alian, Y. Wang, R. Wang, I. Kurakin, C. Tai, and N. S. Kim, "Don't forget the i/o when allocating your llc," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 112–125.

[31] J. Zhang, Y. Liu, G. Jain, Y. Zha, J. Ta, and J. Li, "MEG: A RISCV-based system simulation infrastructure for exploring memory optimization using FPGAs and Hybrid Memory Cube (best paper nominee)," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (**FCCM**)*, April 2019.

[32] J. Zhao, A. Agrawal, B. Nikolic, and K. Asanovic, "Constellation: An open-source soc-capable noc generator," in *2022 15th IEEE/ACM International Workshop on Network on Chip Architectures (NoCArc)*. Los Alamitos, CA, USA: IEEE Computer Society, oct 2022, pp. 1–7. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/NoCArc57472.2022.9911299

[33] J. Zhao, B. Korpan, A. Gonzalez, and K. Asanovic, "Sonicboom: The 3rd generation berkeley out-of-order machine," May 2020.